Parallel Preconditioning and Approximate Inverses on the Connection Machine

Marcus Grote

Dept. of Computer Science
Stanford University
Stanford, CA 94305
grote@na-net.stanford.edu

Horst D. Simon

NASA Ames Research Center
Mail Stop T045-1
Moffett Field, CA 94035
simon@nas.nasa.gov

Abstract

We present a new approach to preconditioning for very large, sparse, non-symmetric, linear systems. We explicitly compute an approximate inverse to our original matrix that can be applied most efficiently for iterative methods on massively parallel machines. The algorithm and its implementation on the Connection Machine CM-2 are discussed in detail and supported by timings obtained from real problem data.

1 Introduction

The solution of large sparse linear systems of equations arises quite frequently as the most important computational kernel in a large number of scientific applications. If the underlying application is based for example on a finite element or finite volume method then the coefficient matrix will usually be unstructured general sparse. For computational fluid dynamics applications involving unstructured meshes [3, 9], and for structural analysis applications with inhomogenous materials the coefficient matrix will also be unsymmetric.

Solution methods for such large, unstructured, unsymmetric, general sparse methods based on iterative solvers usually involve a preconditioning phase, which is designed to improve the convergence of the iterative solver. Up to today, preconditioning methods on highly and massively parallel systems have faced a major difficulty. The most successful preconditioning methods in terms of accelerating the convergence of the iterative solver such as incomplete LU factorizations are notoriously difficult to implement on parallel machines for two reasons: (1) the actual computation of the preconditioner is not very floating-point intensive, but requires a large amount of unstructured communication, and (2) the application of the preconditioning matrix in the iteration phase (i.e. triangular solves) are difficult to parallelize because of the recursive nature of the computation. Triangular solves on parallel machines usually require a reordering of the problem in the form of a wavefront approach, e.g. [1, 8].

Here we present a new approach to preconditioning for very large, sparse, unsymmetric, linear systems, which avoids both difficulties. We explicitly compute an approximate inverse to our original matrix. This new preconditioning matrix can be applied most efficiently for iterative methods on massively parallel machines, since the preconditioning phase involves only a matrix-vector multiplication, with possibly a dense matrix. Furthermore the actual computation of the preconditioning matrix has natural parallelism. For a problem of size n, the preconditioning matrix can be computed by solving n independent small least squares problems.

The new algorithm thus exhibits a natural parallelism, which can be easily exploited on massive parallel SIMD machines. In this paper only the preconditioning algorithm and its implementation on the Connection Machine CM-2 are discussed in detail. The complete linear equation solver and a comaprison of the performance on different machines will be reported in a future report[6].

2 Preconditioning Based on Approximate Inverses

We shall consider the numerical solution of very large but sparse linear systems of the form

$$Ax = b,$$

$$x, b \in R^n$$

without assuming any special properties for A such as symmetry or definiteness. The order of A typically lies between, say, one thousand and one million. However, A usually has just a few nonzero elements per column.

Direct methods based on the LU factorization of A, generally produce fill-in so that L and U might need to be stored on peripheral storage and the I/O costs tend to consume most of the execution time. Iterative methods on the other hand do not suffer from fill-in, and with effective preconditioned and accelerated methods one may derive very efficient algorithms such as GMRES, QMR, etc. — an excellent survey can be found in [2].

Our aim is to construct an approximate inverse M to A and consider applying the iterative solver to the *preconditioned* system

$$AMy = b, \quad M \approx A^{-1}, \quad x = My \; .$$

Set t_s to be the execution time per step without preconditioning, and T_p the total execution time with preconditioning. Roughly speaking,

$$T_p = t_1 + n_p \times (t_s + t_m),$$

where t_1 is the time spent initially to compute the approximate inverse M, t_m the time necessary to compute Mv for some vector v and n_p the number of iterations. In the same way, the total time without preconditioning might be expressed by

$$T_s = n_s \times t_s \; .$$

The "art of preconditioning" consists now in finding a matrix M which minimizes the ratio T_p/T_s.

From the expression for T_p we can immediately deduce what kind of conditions M must satisfy to be effective:

- The crucial time which needs to be minimized is of course t_m, since it occurs at each iteration, i.e. Mv must be very fast on the underlying architecture.

- Since n_p depends heavily on the closeness of M to A^{-1}, it is often worthwile to spend a little more time in the computation of M, than merely apply diagonal scaling for instance. Hence, if the number of iterations n_p is reduced to $\tilde{n}_p$, the total execution time will be lowered to $t_1 + \tilde{n}_p \times (t_s + t_m)$. However, we must keep in mind that t_m might increase for a more sophisticated approximate inverse.

We shall describe one particular choice for M which seems to satisfy these two main requirements in an optimal way for the Connection Machine CM-2.

3 Overview of the algorithm

The CM-2 is a massively parallel SIMD machine, and can, as our preliminary timings show, compute Mv extremely rapidly when M is a banded matrix. Thus it seems natural to require that M be a matrix with , say, $2p+1$ diagonals, $p \geq 0$, so that we may keep t_m very small relatively to the time spent for computing Av. The question is now, how should we compute a matrix M close to A^{-1}, and 'close' in what sense?

The closeness might be measured in some norm $\| \|$, so that we need to find an M which minimizes

$$\|AM - I\|.$$

In general, this problem is even harder than solving $Ax = b$; the main idea of our approach is to choose the norm to be the Frobenius norm, so that the problem decouples, i.e. the columns of M, denoted by M_k, can be computed independently and in parallel.

Let us denote by A_k the columns of A. It is relatively simple to realize that to minimize

$$\|AM - I\|_F,$$

we need to minimize

$$\|AM_k - e_k\|_2$$

for each k individually, $1 \leq k \leq n$ (see [7]). Since each M_k contains at most only $2p+1$ nonzero elements, we need to solve n independent least square problems of size only $n \times (2p+1)$. This can be done by constructing, factorizing and solving all the normal equations simultaneously with parallelism of order n, where n is the dimension of the original system. To construct the normal equations, we must compute the sparse inner products of each column A_k with itself and its $2p$ *nearest* neighbors.

We do not discuss any theoretical results concerning the effectiveness of the preconditioning approximate inverse in the general case. The algorithm has been motivated by a paper by Demko et al. [4], which shows that the inverse of a sparse matrix has entries exponentially decaying away from locations which are nonzero in the original matrix.

4 Preliminary timings

We shall present some preliminary timings computed on two different CM-2's: an 8K processor machine at Stanford University, and a 32K processor machine located at NASA Ames Research Center. We shall pursue one particular choice for implementing the sparse inner products (SPIP), which has the advantage of necessitating only a minimum amount of memory.

So far we have tested only the preconditioning phase on a number of test matrices from the Boeing-Harwell collection. Some of the results of our tests and their performance on the CM-2 are given below. We have also implemented the algorithm on the Cray Y-MP and reached complete agreement in the numerical accuracy of the computed preconditioning phase.

In this section we focus on very specific and small instruction groups. Let M be an $n \times n$ real matrix with p nonzero diagonals, p odd. We now compare three different ways of multiplying M with some vector x:

a) Using the general sparse matrix/vector multiply routine supplied by the CMSSL library. One sparse-matvec-setup is done initially, and not taken into account for the timings.

b) Using a customized multiply that involves $2(p-1)$ CSHIFT operations, M being stored by colums.

c) Same as b) where M is stored by rows, so that only $(p-1)$ CSHIFT operations are necessary.

Average time in seconds with $p = 3$ for 100 mat/vector multiplications on a 4K machine is given below

n	1K	4K	16K	32K
a)	0.6362	1.468	5.195	10.11
b)	0.0348	0.0419	0.0871	0.148
c)	0.0228	0.02549	0.05922	0.104

Average number of MFlops for the same numerical experiment is given below

n	1K	4K	16K	32K
a)	0.80	1.4	1.57	1.62
b)	14.7	48.9	94.1	111
c)	22.4	80.3	138	157

From these results it is obvious that one should never use the general sparse matrix/vector multiply routine when the matrix is tridiagonal, since it is almost a hundred times slower. Let us now take a closer look at option c) for different values of p .

Average time in seconds and MFlops with $p = 3, 5, 7, 9$ for 100 mat/vector multiplications on a 4K machine are given below, $n = 32K$:

p	3	5	7	9
time	0.1045	0.1773	0.2776	0.3458
MFlops	157	166	153	161

During these extensive timings, we noticed some additional interesting behaviors:

The first version of the sparse matrix/vector multiplication was written in such a way to accept p as a parameter and then by means of calculated indices to perform the cshift's and the multiply's in do loops. This version was taking an additional 70% amount of time than the second and more optimized version, where for each value of p the appropriate instructions were separately coded.

We remark also that whenever M is stored by colums like under b), it can easily be transposed and then used as under c) at relatively cheap cost. Again we believe that a transposition routine minimizing communication costs for each value of p separately would probably yield the best results.

The overhead due to setting up the multiplication into a subroutine is negligible.

We executed the same program on a different 8K processors Connection Machine and got the following results. Compilation was always done under the slice-wise option.

Average time in seconds and MFlops with $p = 3, 5, 7, 9$ for 100 mat/vector multiplications on a 8K machine are given below, $n = 32K$:

p	3	5	7	9
time	0.0597	0.104	0.158	0.205
MFlops	275	284	270	272

Average time in seconds and MFlops with $p = 3, 5, 7, 9$ for 100 mat/vector multiplications on a 32K machine are given below, $n = 32K$:

p	3	5	7	9
time	0.04064	0.06632	0.08573	0.11390
MFlops	403	445	497	489

It should be noted here that all these timings are subject to fluctuations which sometimes can be far off.

In addition, we present some timings that demonstrate the effectiveness of the CM-2 for some crucial, massively parallel, low-level operations. We consider real arrays of length 32K and perform both 100 circular shifts and 100 Hadamard products (element by element multiplications) on processor arrays of increasing size. So that for $n = 32K$,

processors	4K	8K	32K
time for CSHIFTS	0.0173	0.0122	0.0088
time for Hadam. Prod.	0.0319	0.0156	0.0048

As a general remark we may say that the execution time of a CSHIFT operation is almost processor independent; the Hadamard product's execution time, however, decreases linearly with increasing number of processors. This can be explained as follows: when we increase the number of processors from say 4K to 32K, parallelism for the CSHIFT increases; however, communication costs do increase as well, since in the 4K case, a CSHIFT of part of a vector residing on one processor doesn't involve any "physical" communication at all. In the Hadamard product case, no communications are necessary, so that increasing parallelism scales up very well.

5 Sparse Parallel Inner Product (SPIP)

In fairly many applications, one encounters the following problem. Given two sets $V1$ and $V2$ of n vectors each, we would like to compute the inner product of the ith vector in set $V1$ with the ith vector in set $V2$. Of course this problem can be easily fully parallelized on a massively parallel machine such as the CM, even for very large n. This at first seemingly trivial problem, becomes far more difficult when the vectors considered are supposed to be **sparse.**

One commonly used approach is to scatter the packed vectors into full length vectors of size n, and then to perform the inner products on the full vectors, regardless of the sparsity pattern. This creates a large overhead in the computation, since most of the operations involve zeros. Although this approach can be very interesting on a vectorizing computer, where the inner product is extremely fast, it is prohibitive on the CM because of the relatively low performance of the CM processors in floating point arithmetic. Even though this last issue is debatable, the bottom line is that for very large n, say 32K, there is simply not enough physical memory on the CM-2, since 32K real vectors of length 32K take 4 Gigabytes of memory.

Let us now present a different approach that does not demand either computation overheads or enormous memory space allocations. We must always keep in mind that the CM is a SIMD machine, and since the sparsity structures of the vectors are supposed to be totally different, that one first "simple-minded" implementation might be merely rejected by the compiler.

5.1 Description of the SPIP algorithm

We give a complete and detailed description of the algorithm, emphasizing features particular to the CM. We suppose that an upper bound MAX to the maximum number of nonzero elements per vector is known. Now the ith vectors of both sets $V1$ and $V2$ are laid out on the ith processor along the serial axis. Each vector is described in the obvious way by two arrays of MAX length. The first one, V1(j,i), is a real array which contains all the nonzero elements of the ith vector of V1, eventually patted with zeros at the end. The second one, I1(j,i), is an integer array which contains the row indices and hence describes the sparsity structure. For instance, if A were an $n \times n$ sparse matrix stored by colums, then $A(i,j) = V1(I1(i,j),j)$.

All the following variables but $V1, V2, I1, I2$ are one dimensional arrays laid out along the parallel (:NEWS) axis along the processor array. The resulting inner products are returned in P, and fully computed in parallel. Furthermore, we define two arrays $IX1$ and $IX2$ which contain the index j for the next element to be considered in each vector.

step 1 Initialize $IX1$ and $IX2$ to 1, and P to zero.

step 2 Copy the actual row index of the next element to be considered in each vector into temporary arrays $LOW1$ and $LOW2$. (This step is necessary because CM FORTRAN does not allow arithmetic operations on variables such as $A(INDEX(I), I)$, and is done efficiently — according to the CM-FORTRAN User's Guide— by the CMF_AREF_1D subroutine.)

set 3 Compute a logical $MASK$, true whenever $LOW1$ is equal to $LOW2$.

step 4 Copy the actual real value of the next element to be considered in each vector into temporary arrays $VL1$ and $VL2$.

step 5 Where $MASK$ is true, compute $P = P + VL1 * VL2$, and increment $IX1$ (and $IX2$, actually done under step 6).

step 6 Where $LOW1$ greater **or equal** to $LOW2$ increment $IX2$, elsewhere increment $IX1$.

step 7 If any $(VL1 \neq 0 \wedge VL2 \neq 0)$ goto step 2.

One iteration uses at mosts $2n$ flops computed in parallel, i.e. at most 2 flops per processor. It can be easily shown that this algorithm terminates after

at most MAX iterations, and takes at least MAX iterations (for consistent data, i.e. MAX not overestimated, etc.). In the actual implementation, the serial dimension is MAX+1 instead of MAX; thus, we introduce a memory allocation overhead of n (1 floating number per processor), to avoid additional testing. However, this does not create any computational overhead: the algorithm performs at least on one processor the absolute minimum number of multiplications necessary to compute the inner product.

5.2 Performance analysis

We applied a first implementation to 38K sparse vectors of full length 38K, with one thousand nonzero elements in each, on a 32K CM. The execution time was 1.24 seconds in the best, and 2.48 seconds in the worst case. In the best case, the vectors in $V1$ had the same sparsity structure as the vectors in $V2$. In the worst case, the positions of the nonzero elements in $V2$ were shifted by one with respect to the nonzero elements in $V1$. These timings confirm of course the intuitive idea that the execution time should be exactly proportional to the number of (identical) iterations executed serially. Since the worst case takes twice as many iterations (2 MAX) as the best case, it also takes twice as long to execute. More timings confirmed that the execution time is exactly proportional to MAX (for a same sparsity structure), which again is obvious for similar reasons.

6 The complete algorithm

In this section we give a precise description of the algorithm and introduce some notations.

Let A be an $n \times n$ real sparse matrix. We denote by A_k the kth column of A. Let MAX be the maximum number of nonzero elements per column. To each column A_k we associate a row index vector r_k so that $a_{ij} = A_j(r_j(i))$. Since A is sparse, we only store its nonzero elements, so that for each k, A_k and r_k are of length MAX, padded with zeros at the end if necessary. In the following, assume that A is initially stored by columns, i.e. that for each k, A_k and r_k reside on the same processor. We allocate as well a real work array W_k of the same size as A_k with some row index array.

Denote by M the approximate inverse of A, M_k its kth column. We impose a banded structure upon M and denote its bandwidth by p, i.e. M has $2p+1$ nonzero diagonals. Each M_k — which are the unknowns — resides on the same processor as A_k; M_k is a $2p+1$ real vector, $M_k(i) = m_{(k+p+1-i)k}$, padded with zeros when either $i \geq n+p+2-k$ or $i \leq p+1-k$, $1 \leq i \leq 2p+1$.

Consider now the $(2p+1) \times (2p+1)$ real matrices B_k, $1 \leq k \leq n$, where

$$B_k(i,j) = A^{\top}_{k-p-1+i} A_{k-p-1+j} \ , 1 \leq i,j \leq 2p+1$$

B_k is padded with zeros where $i \leq p+1-k$, $i \geq n+p+2-k$, $j \leq p+1-k$ or $j \geq n+p+2-k$, unless $i = j$ since then we set $B_k(i,j) = 1.0$ so that the B_k be nonsingular. Obviously the B_k are symmetric and positive definite, so that we need only compute and store their lower part.

Finally we define the real vectors C_k of length $2p+1$, $C_k(i) = A_{k(k-p-1+i)}$, i.e. the $2p+1$ elements in the kth row of A (now seen as a dense matrix) clustered around the diagonal. Again, C_k is padded with zeros when $i \leq p+1-k$ or $i \geq n+2+p-k$, $1 \leq i \leq 2p+1$.

The full scheme can be outlined as follows:

- Construct the matrices B_k
- Factorize $B_k = L_k D_k L_k^{\top}$
- Solve $L_k y_k = C_k$, $D_k z_k = y_k$, and $L_k^{\top} M_k = z_k$

We now discuss each phase individually, assuming that each kth processor contains $A_k \in R^{2MAX}$ for $1 \leq k \leq n$, and $C_k \in R^{2p+1}$.

6.1 Construct the matrices B_k

This is a critical phase, since it is the only one which involves inter-processor communications which could lower the performance considerably. We recall that the matrices B_k contain the inner products of the A_{k-p} through A_{k+p} columns. It is important to realize that B_k's $k+j$th row is equal to B_{k+j}'s kth row, for $-p \leq j \leq p$. In addition we need two small work arrays P_k and Q_k of size $2p+1$ each as before laid out through the processor array.

We start by setting $P_k(1) = A_k^{\top} A_k$, which is easily computed in parallel without any conditional statement, and set $W_k = A_k$. Now, for $l = 2, ...2p+1$ do W_k = CSHIFT(W_k, DIM=2, SHIFT=1), so that $Wk = A_{k+l-1}$, and set $P_k(l) = A_k^{\top} W_k$ end do — of course the row indices were shifted as well. Basically this previous loop performs the sparse inner product of each A_k with $A_{k+1}, ...A_{k+2p}$. Now save P_k in Q_k for future purposes. All the entries in the B_k have been computed, but they still need to be shifted to the p left and right neigbors, since they all overlap. This can be done in a quite tricky way, in order to minimize

communication costs. It should be noted that the following communications involve only vectors of length $2p+1$, in contrast to the previous loop, where the vectors were of length 2MAX. First we shift P_k to all the right neighbors: for $j = p$ downto 1, P_k=CSHIFT(P_k, SHIFT=-1), $B_k(1, j : 2p+1) = P(1 : 2p+2-j)$ end do. Similarly for the left neighbors using Q_k, for $j = p+2$ to $2p+1$, do Q_k = CSHIFT(Q_k,SHIFT=1), $B_k(j, j : 2p+1) = Q_k(1 : 2p+2-j)$, end do.

Since we are performing CSHIFT operations, we must take into account wrap-around effects. To avoid destroying values at the ends of the array, we simply allocate $2p$ additional columns for A which are set to zero initially. An alternative approach, where we considered the EOSHIFT instead of the CSHIFT command, was quickly abandonned because of the prohibitive slowness of the EOSHIFT command (same effect as CSHIFT without wrap-around).

6.2 $LDL^\top$ decomposition of B_k

Since all the B_k are symmetric and positive definite, we can factorize $B_k = L_k D_k L_k^\top$. As an alternative, we take a look at the $GG^\top$ Cholesky factorization which requires $2p+1$ square roots, and compare operation counts for Algorithms 4.1.2 and 4.2.1 [5]; here n is the dimension of the system which in our application $(= 2p+1)$:

Opcounts for $LDL^\top$

Op	Factor	Solve	Total
+	$\frac{n^3-n}{6}$	n^2-n	$\frac{n^3+6n^2-7n}{6}$
	$\frac{n^3+6n^2-7n}{6}$	n^2-n	$\frac{n^3+12n^2-13n}{6}$
/	n	n	$2n$
$\sqrt{x}$	0	0	0

Opcounts for $GG^\top$

Op	Factor	Solve	Total
+	$\frac{n^3-n}{6}$	n^2-n	$\frac{n^3+6n^2-7n}{6}$
	$\frac{n^3+3n^2+2n}{6}$	n^2-n	$\frac{n^3+9n^2-4n}{6}$
/	n	$2n$	$3n$
$\sqrt{x}$	n	0	n

Additional timings, performed on a 4K machine upon vectors of length 4K, showed that square roots are 20% slower than divisions, which are twice as expensive as additions or multiplications. More precisely, if x and y are two vectors of length 4K laid out along a 4K :NEWS array, then time($x*y$)=time($x+y$), time(x/y)= 2 time($x+y$) , and time($\sqrt{x}$)=2.4 time($x+y$). Thus we decided to go for the $LDL^\top$ factorization algorithm, as described in [5] under Algorithm 4.1.2 , since it involves fewer divisions, no square roots, and a comparable amount of additions and multiplications. We expect a very high performance, since no communications are necessary during this phase.

6.3 $L_k y_k = C_k$, $D_k z_k = y_k$, and $L_k^\top M_k = z_k$

This final step is straightforward, and again we use the standard forward and back substitution algorithms 3.1.1 and 3.1.2 [5] (see the previous section for operation counts).

7 Extensive timing data

We consider three matrices taken from the well-known Boeing-Harwell collection, which arise in black oil simulators: SHERMAN1, SHERMAN2 and SHERMAN5.

For the SHERMAN1 black oil simulator, $n = 1000$, and A contains 3750 nonzero elements:

On an 8K processor CM-2, we obtain:

Results for SHERMAN1

$2p+1$	8K proc		32K proc	
	time	Mflops	time	Mflops
1	0.003	12.7	0.003	11.9
7	0.07	10.2	0.10	7.9
13	0.17	17.1	0.24	12.3
19	0.32	23.3	0.47	16.0
25	0.53	28.6	0.81	18.9
31	0.82	33.0	1.27	21.3
37	1.20	36.7	1.89	23.3
43	1.69	39.5	2.70	24.8
49	2.31	41.9	3.70	26.1

Because of the small size of the matrices, not only do we get very low performances, but the program runs faster on the smaller 8K processor machine than on the full 32K processor one. This again comes from the fact that the 32K full processor array is merely too large for these small matrices.

For the SHERMAN2 black oil simulator, $n = 1080$, and A contains 23094 nonzero elements:

Results for SHERMAN2				
$2p+1$	8K proc		32K proc	
	time	Mflops	time	Mflops
1	0.003	44.0	0.004	40.3
7	0.25	6.1	0.30	5.1
13	0.53	8.2	0.65	6.8
19	0.88	10.9	1.10	8.7
25	1.29	13.9	1.68	10.7
31	1.78	17.2	2.38	12.9
37	2.36	20.5	3.23	14.9
43	3.04	23.6	4.25	16.9
49	3.85	26.5	5.48	18.6

We notice here, like in all other examples, that there is a big 'jump' from $2p+1=1$, ie $p=0$ and M is diagonal, to $2p+1>1$. This is because in the case of $p=0$, neither nearest neighbor communications (CSHIFT instructions) nor indirect addressing (CMF-AREF-1D instruction) is necessary; thus everything is purely local and very fast.

For the SHERMAN5 black oil simulator, $n=3312$, and A contains 20793 nonzero elements:

Results for SHERMAN5				
$2p+1$	8K proc		32K proc	
	time	Mflops	time	Mflops
1	0.004	44.2	0.003	57.3
7	0.19	11.1	0.11	18.4
13	0.43	16.2	0.25	27.4
19	0.74	22.3	0.43	38.1
25	1.14	28.5	0.66	49.2
31	1.67	34.1	0.95	59.7
37	2.32	39.3	1.31	69.6
43	3.14	43.8	1.76	78.0
49	4.12	47.8	2.29	81.9

Here we discover on one hand that the 32K processor machine performs better than the 8K smaller one, and on the other hand, that the MFlops rate is almost doubled. This is now clearly due to the fact that the size of the matrix has increased, so that we start making use of the massively parallel power available on the CM-2. Therefore we believe that for much larger problems of the order of, say, 100K, our algorithm would yield very high performances.

We anticipate to solve problems of order up to 60,000 on the CM-2 by the time of the conference. It should be mentioned that for matrices in the range of sizes ranging from about $n \approx 1,000$ to $n \approx 10,000$ the performance on a single processor of the Cray Y-MP is about 30 Mflops, which appears to be the asymptotic limit for the Cray Y-MP.

8 Conclusions

We have addressed the problem of preconditioning very large but sparse matrices of general structure on the Connection Machine CM-2. We pointed out that the main issue is not the pre-computation of the preconditioner, but the speed at which it must be applied within the iterative solver at each step. One optimal choice for the CM-2 is to compute an explicit, banded, approximate inverse to our original matrix, which may be applied close to the machine's peak performance. We presented an algorithm and its implementation on the Connection Machine; additional timings make us believe that even the pre-computation — not only the application — of the preconditioner would be extremely efficient for general, very large systems. We hope to support our expectations by numerical experiments in the near future.

9 Acknowledgements

We would like to thank F. Alizadeh, O. Ernst, G. Starke, Z. Johan, B. Biondi, and Prof. G. Golub for their interesting and very helpful comments.

References

[1] E. Anderson and Y. Saad. Solving sparse triangular linear systems on parallel computers. *Int. J. High Speed Computing*, 1(1):73–96, 1989.

[2] O. Axelsson. A survey of preconditioned iterative methods for linear systems of algebraic equations. *BIT*, 25:166 – 187, 1985.

[3] T.J. Barth. On unstructured grids and solvers. In *Computational Fluid Dynamics, Lecture Series 1990-03*. Von Karman Institute, Belgium, March 1990.

[4] S. Demko, W. F. Moss, and P. W. Smith. Decay rates for inverses of band matrices. *Math. Comp.*, 43(168):491 – 499, 1984.

[5] G. Golub and C. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 1983.

[6] M. Grote and H. Simon. Approximate inverse preconditioners. (Technical Report in preparation, 1992).

[7] L. Yu. Kolotilina and A. Yu. Yeremin. Factorized sparse approximate inverse preconditionings. 1990.

[8] J. Saltz. Automated problem scheduling and reduction of synchronization delay effects. Technical Report 87-22, ICASE, July 1987.

[9] V. Venkatakrishnan and D. Mavriplis. Implicit solvers for unstructured meshes. Technical Report RNR-91-16, NASA Ames Research Center, Moffett Field, CA 94035, April 1991.